THE "INSTANT SEQUENCING" TASK: TOWARD CONSTRAINT-CHECKING A COMPLEX SPACECRAFT COMMAND SEQUENCE INTERACTIVELY

Joan C. Horvath, Leon J. Alkalaj, Karl M. Schneider, and Arthur V. Amador

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
USA

N94-23875

Joseph N. Spitale
California Institute of Technology
Pasadena, CA 91125
USA

ABSTRACT

Robotic spacecraft are controlled by sets of commands called "sequences." These sequences must be checked against mission constraints. Making our existing constraint checking program faster would enable new capabilities in our uplink process. Therefore, we are rewriting this program to run on a parallel computer. To do so, we had to determine how to run constraint-checking algorithms in parallel and create a new method of specifying spacecraft models and constraints. This new specification gives us a means of representing flight systems and their predicted response to commands which could be used in a variety of applications throughout the command process, particularly during anomaly or high-activity operations. This commonality could reduce operations cost and risk for future complex missions. Lessons learned in applying some parts of this system for the TOPEX/Poseidon mission will be described.

Key Words: Sequencing, mission operations, automation, parallel computing.

1. INTRODUCTION

Robotic spacecraft are controlled by sets of time-tagged onboard commands called "sequences." These sequences must be verified and checked against mission constraints to be certain that they will have the planned effect on the spacecraft. One of the key automated portions of the sequence verification process is a checking program which does some limited functional simulation of the sequence running on the spacecraft and compares the resulting spacecraft states against mission rules. Constraint-checking occurs at many points in the so-called "uplink process", which creates spacecraft command sequences. Checking occurs at the command timing interaction level, but also at the high-level planning and sequence-integration stages.

Making our existing sequence-checking program, (called the "Checker") faster would make some changes in our uplink process, since then people building command sequences would not tend to check a sequence by hand before submitting it to lengthy batch runs. We determined that a good way to make the Checker code run faster would be to run it on a parallel computer, which meant that we had to determine how to run constraint-checking algorithms in parallel. Directly porting the existing Checker code to a parallel machine, however, proved to be difficult due to the inherently non-parallel way in which the flight rules and the spacecraft models on which they acted were encoded [Refs 1-2.] Once we realized this, we concentrated our efforts on exploring better ways of specifying constraints and models.

When we created this new specification system, we realized that we had a way of describing flight systems that was more broadly applicable than just within the traditional "sequence checking" part of the flight command process. A general representation of flight systems and their predicted response to commands could be used in a variety of applications throughout the uplink process, particularly during anomaly or high-activity operations. We currently have two development efforts under way: a system intended to run on a parallel computer, and an operational (sequential) system for use by the TOPEX/Poseidon spacecraft.

2. SPECIFICATION AND VERIFICATION

2.1. Basics

We describe spacecraft systems in terms of three

fundamental pieces: the rules to be checked, a model of the subsystem(s) of the spacecraft or ground system to which the rule(s) apply, and an "action table" that describes interactions among the models and illegal transitions inside the models [Ref. 3]. It is also very desirable to have "status events" that flag, in user-readable form, changes of state of some spacecraft components. Each of these is now described in turn.

2.2 Rules

When a spacecraft is designed, "flight rules" or "mission rules" are developed to prevent damage to the hardware, to prevent loss of science data, or to simplify and constrain the operation of the spacecraft.

In our system, which we call SAVE (Specification and Verification Environment) each rule to be checked by software is expressed as a logical constraint over state transitions. The constraint can be of temporal nature as well as a constraint over state orderings. Informally, the syntax of a rule is as follows.

Whenever *(a state -> a certain value)*
 if *(some condition holds)*
 =>(a violation of the flight rule has occurred).

Where the "->" symbol should be read as "goes to" and the "=>" value should be read as "generates." The states and models used in the rules are defined in *models* and *action tables.*

2.3 States

The states may be in the format

model.state

where ***model*** is the name of the model to which the state belongs, and ***state*** is the particular state variable. For temporal comparisons, the state may also be shown as:

model.state.time

where the ***time*** field is the time at which a given state achieved its most recent value. The syntax used inside the "if" clauses of the rules is the standard syntax for a logical expression in the "C" programming language; e.g. "&&" for "AND", "||" for "OR", and so on.

2.4. Status information

It is desired that the status of certain variables be printed out whenever the state changes (even if the change is not a flight rule violation.) The syntax for status events is similar to that for rules, namely:

Whenever *(state-a goes to a value)*
 if *(condition)*
 then status *(state-a,state-b,state-c....)*

where *state-a* is the "trigger state" which will cause a status message to be generated, and *state-b, state-c,* ... are states the user may want to see as well when state-a changes. Usually the condition in the "if" statement will be TRUE unconditionally -- that is, the status event will always be printed out irrespective of any other states.

2.5 Models

Every rule and status event requires that one or more "models" of a limited subset of the spacecraft behavior be generated. These models can be shown in "finite-state" form: that is, a portion of the spacecraft is modeled in terms of several discrete variables (an "A" and "B" redundant side, for example). The commands or other actions that cause the system to transition from one state to another are shown on the arcs between the commands. [Figure 1].

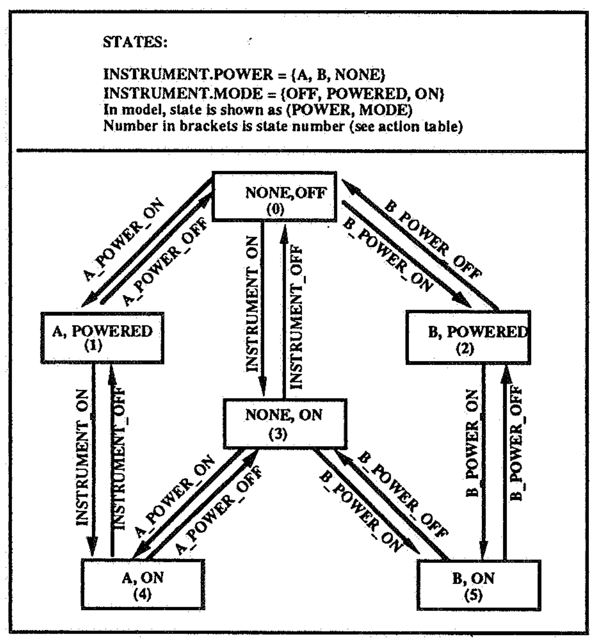

Figure 1. A typical state diagram.

Figure 1 models an instrument on board a spacecraft which can be powered by one of two redundant power supplies - A or B. The command to close the relays from the instrument to the A power supply is the "A_POWER_ON" command; similarly with the "B_POWER_ON" command. Once the instrument is powered, it can be turned on to its operational mode with the "INSTRUMENT_ON" command. This model shows that the instrument can be turned on when it is unpowered, but it will then surge when power is applied This "illegal transition" is discussed in the following section.

2.6 Action tables

The finite-state models can also be shown more completely in a spreadsheet-like form. This form has all the legal states in the vertical direction, and all the transition commands in the horizontal direction Every command is analyzed for its actions should it be issued in any of the states. In some cases this will work out to nothing happening; in others this command will cause an illegal transition. These illegal transitions will also be flagged by the checking software if and when they occur. The "next state" field in each box of the action table tells the software what the next state would be should the relevant command be issued.

Figure 2 shows an "action table" for the system shown in Figure 1. The blank boxes in the Action rows imply that the transition simply takes place to the "Next State" shown and no side effects occur. The "ERROR" states shown (for example, the transition from State 0 to State 3 caused by the INSTRUMENT_ON command issued when the spacecraft is in State 0) would generate an "illegal transition" message to the user. Arbitrarily complex "side effects" (e.g., effects on other models) can appear in these "action" boxes in the form of function calls in the "C" programming language.

Let's say that there is a restriction on the system shown in Figures 1 and 2 that will not allow other instruments to be turned on if this instrument happens to be turned on. A typical rule, then, for the system described in Figure 1 might then be expressed as:

Whenever (INSTRUMENT.MODE -> ON)
if (OTHER_INSTRUMENT.MODE == ON)
*=> **violation.***

Note that there is no hard-and-fast distinction between an "illegal transition" and a "flight rule"; some illegal transitions have been called out as particularly troublesome and made rules. Implementationally, some are easier to call out one way and some in others.

3.0 PARALLEL IMPLEMENTATION

3.1. Overview

Our next step was to build a small prototype system that used these principles and implemented them on a parallel computer. This SAVE software implementation consists of two major functions. The primary function is sequence verification; the second (related) function is system specification. System specification is a kind of "installation" function which normally would not be used routinely. In system specification, an operator using finite state machine notation lays out a spacecraft/ground model, and specifies constraints upon the behavior of this model. System verification is "production" running of the system to check a sequence with pre-defined constraints.

The system was designed in three basic parts: the "core", the "compiler" and the "database." [Figure 3]. The compiler takes as input rules in the "whenever" syntax described above, and models in a database.

NO.	POWER	MODE	COMMAND->	A_POWER_ON	B_POWER_ON	A_POWER_OFF	B_POWER_OFF	INSTRUMENT_ON	INSTRUMENT_OFF
0	NONE	OFF	ACTION					ERROR	
			NEXT_STATE	1	2	0	0	3	0
1	A	POWERED	ACTION		ERROR				
			NEXT_STATE	1	1	0	1	4	1
2	B	POWERED	ACTION	ERROR					
			NEXT_STATE	2	2	2	0	5	2
3	NONE	ON	ACTION	ERROR	ERROR				
			NEXT_STATE	4	5	3	3	3	0
4	A	ON	ACTION		ERROR	ERROR	ERROR		
			NEXT_STATE	4	4	3	3	4	1
5	B	ON	ACTION	ERROR		ERROR	ERROR		
			NEXT_STATE	5	5	5	3	5	2

Figure 2. An Action Table.

format derived from the action table format shown in Figure 2. When a user adds a model or rule, these inputs are compiled into in-line C code, which is linked with the core code.

3.2. Runtime environment

At runtime, the core code determines from a user input how many processors and models it will have for the given run. It assigns the models to processors according to a user-specified definition. A sequence of commands to be checked against rules is read and stripped of all information that is not relevant to the sequence rule checking function. The commands in the sequence are then partitioned out to the different processors according to the model to which they "belong."

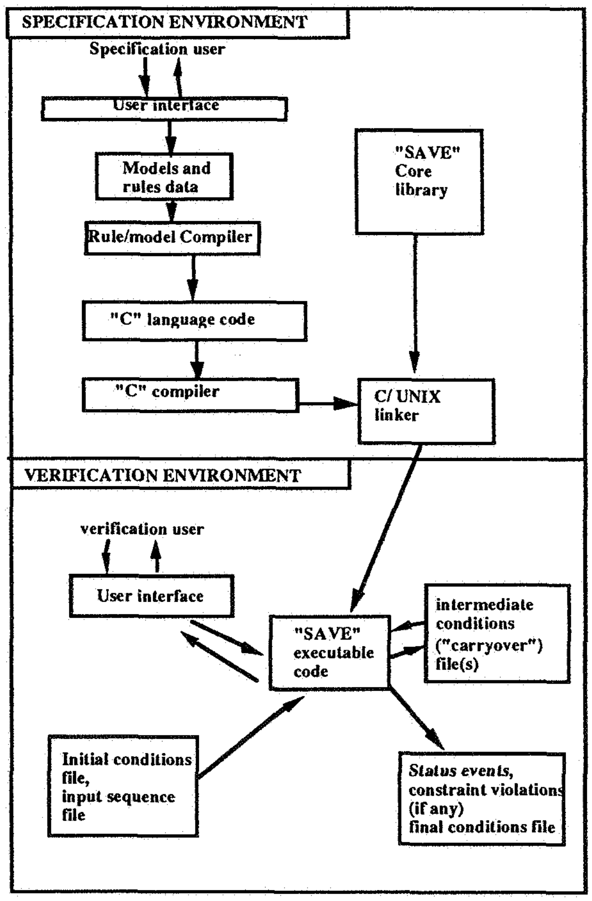

Figure 3. Architecture summary.

Then, starting from an initial state for each model, each command's effects on the system are simulated according to the data in the action tables. If a state changes that triggers a rule check, the rule is activated and an error message, if any, is output to the error log. A final state is written to a file at the end of the run.

3.3. Parallel processing results

A prototype was built on a five-processor parallel computer. In this prototype, a "conservative" synchronization approach [Ref.4] is used to ensure that processors do not get out of synchronization with each other. This prototype showed that this methodology could achieve excellent speedup for this type of problem depending heavily upon the actual models and sequences being checked. (Models which interact with each other frequently make less efficient use of the parallel computer. Sequences which use one model disproportionately make less efficient use, as do very short sequences where work tends not to average out over the parallel processor too well.)

The methods used in the parallel prototype could easily be adapted to a distributed system or to any of a variety of parallel architectures. The system is best suited for the type of parallel processor which has substantial memory and processing power on each node (i.e., a "large grained" or "medium grained" machine). We also use this parallel system as the basis for our one-processor system we are implementing for the TOPEX/Poseidon project. For the one-processor version, we have built routines that mimic the parallel interprocessor communications and return the same value(s) as would the parallel one. Since most of these routines do nothing on one processor, this was not a complex task.

4.0 LESSONS LEARNED FROM FLIGHT IMPLEMENTATION

4.1. Moving a prototype into a flight environment

Recently it was decided to build a sequence-checking system for the TOPEX/Poseidon spacecraft based on the SAVE prototype system. This full system is called "MSAVE" (for Mission Planning, Sequencing and Scheduling Specification and Verification Environment.) Several modifications were necessary to the system for it to be used in the TOPEX environment [Ref.5].

4.2. Modifications required

These modifications fell into three basic categories. First, some additional functionality in the core code

was needed to handle TOPEX-specific issues (TOPEX file formats, handling of some special command types, etc). Secondly, a more robust and friendlier user interface was needed than the text-editor-dependent interface of the prototype. A graphical user interface based on X-windows is being developed, heavily re-using code that was developed for the rest of the TOPEX sequencing system. Thirdly, some convenience functions to assist people developing large sequences were added.

One of these "conveniences" was the addition of "intermediate carryover." It is always necessary to carry over state information from the end of one sequence to the beginning of the next. Intermediate carryover, however, is used to allow MSAVE to start a run part-way into a sequence file, similar to the concept of "checkpointing" often used in large computational physics runs. If an anomaly occurs on the spacecraft, this capability can also be used to allow the user to change a state at a given time without a command (should the spacecraft exercise some of its automated fault protection, for example.) This same capability could also be used for "what-if" tests: if a sequence is built and someone in real time wants to know "what happens if I turned off this piece of hardware here with a realtime command'?

4.3. Testing issues

In addition, several issues arose that were not as critical in the research prototyping environment. Primarily these issues were in the realm of compatibility with existing software, testing and verification. The most significant of these was that the MSAVE system is novel among flight sequence checking software in that it allows a user to compile new rules and models into in-line code, thereby reducing the coding complexity and runtime penalties inherent in interpreted code while avoiding the inflexibility of hardcoded models. When a rule or model is added and saved, MSAVE will automatically recompile and relink the relevant MSAVE executables. However, the following issue inevitably arises:

> *"When we add a rule or model and add it to the code and recompile, what retesting is necessary ?"*

The addition of a rule or model is generating code, but the code is machine-generated. This machine-generation process will be exhaustively tested prior to project delivery. The syntax of the code is also limited and predefined. Further, rules and models are isolated from the rest of the processing code, and a tracking scheme to avoid the possibility of overwriting variables or addresses in the rest of the executable has been developed. A syntax and format checking process takes place in MSAVE when a compilation request for a new rule or model is received to enforce this isolation. These limitations should bound the amount of testing that will be required.

A good analogy to the MSAVE testing situation is that when one develops flight code in C it is not seen as necessary to re-regression-test the C compiler for every delivery of the C software in addition to testing the code that has been written. The specification environment portion of MSAVE can be seen as a compiler for the flight rule and model language, and as such can be tested once and then the "programs" (the new rules and models as they are added) can be debugged separately. However, there is a capability in MSAVE to add functions in action tables for complex side-effect calculations. These functions will need to be generated in an editor, and limited functional testing of the system will be performed should a function like this be linked into MSAVE.

4.4. Limitations of the Verification Environment

MSAVE will not be able to detect constraint violations which occur after the end of a sequence, although if a subsequent sequence is checked they will be detected at that time. MSAVE verifies constraints using information available in the command sequence file only. This implies that certain constraints, such as those requiring precision modeling of spacecraft turns, solar array slewing, or orbit position propagation cannot be verified without significantly expanding the complexity and scope of rules and models in this implementation of MSAVE (although the general SAVE methodology in Section 2 could support this functionality).

4.5. Limitations of the Specification Environment

4.5.1. Command Scheduling And Cyclic Graphs

Sometimes it arises that a command has an effect later in time than the command itself; for example, a heater takes a thruster to the "warm" state after some time delay. These effects are modeled by "scheduling" a pseudocommand at a later (or, in some cases, the same) time. This ability to schedule commands is also the essential mechanism by which a given model

can effect changes upon the state of another model. The danger arises from self-scheduling commands, which can lead to a cyclic graph; i.e., Command A which schedules Command B which schedules Command A which .... whenever Command A or B is in the sequence file. This problem is handled in the flight version by prohibiting these loops, but a robust way of handling this situation is of interest for future versions.

4.5.2. Model Modification And Consistency Maintenance

The user will load into memory and edit only one model at any given time. Inconsistencies can arise because of this; for example, if Model B reads a state variable of Model A, and the user decides to delete this state variable in Model A, then Model B now has a reference to an undefined state variable. When the models are compiled, the inconsistency is detected and an error generated. It would be preferable to catch the error as soon as possible, but if MSAVE tried to catch all inconsistencies as each change to a model was entered, its performance would probably be unacceptable to the user if there were more than a few models in the system.

4.6. Experience Encoding Rules

We have now used the rules/action table system to develop and encode a variety of TOPEX flight rules. We have found the system to be a very good and systematic way of doing "knowledge capture" while the spacecraft experts who actually built the hardware are still around, since there is a tendency for a project to slowly lose expertise over time.

The action table format forces one to think through what happens if a command is sent in any of the possible spacecraft states that apply to that command, leading to thorough system behavior specification. It also leads to easier review and discussion than discussion of code in a programming language.

5. CONCLUSIONS AND PLANS

The specification and verification environment described in this paper has been successfully implemented as a prototype on a parallel machine, and is being fully implemented for the TOPEX/Poseidon project on a single-processor workstation. The environment has proven useful in generating real rules and models. When the full TOPEX implementation and rule/action table set is available, we will take this large set and a set of sequences and determine the parallel efficiency of this full implementation. This will give us a basis for extrapolation for other missions and applications.

6. ACKNOWLEDGEMENTS

The authors would like to thank the JPL Director's Discretionary Fund and the TOPEX/Poseidon project for their interest in and support of this work. We also would like to recognize the contributions to the TOPEX-specific implementation by JPL's Dennis Page, Carlos Carrion and Robert Gustavson. Dr. Arkady Kanevsky of Texas A& M University, Dr. Krishna Kavi of the University of Texas at Arlington, and Dr. James Peters of the University of Arkansas had many helpful discussions with the authors during their tours as NASA/ASEE Summer Faculty Fellows at JPL. Joe Spitale was a participant in the Caltech Summer Undergraduate Research Fellowship program while working on this project. The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

7. REFERENCES

1. Horvath, J.C. and Perry, L.P., "Hypercubes for Critical Spacecraft Command Verification." AIAA-90-5095, In *Proceedings of AIAA/NASA 2nd International Symposium on Space Information Systems*, September 17-19, 1990, Pasadena CA.

2. Horvath, J.C., Tang, T., Perry, L.P., Cole, R.C., Olster, D.B. and Zipse, J.E., "Hypercubes for Critical Space Flight Command Operations." In *Proceedings of The Fifth Distributed Memory Computing Conference*, Charleston, SC, April 8-12, 1990.

3. Alkalaj, L.J., "Towards a Specification Language and Programming Environment for Concurrent Constraint Validation of Spacecraft Commands." JPL Internal Report, July 1992.

4. Chandy, K.M., and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations." Communications of the ACM, Vol 24, No. 11, April 1981.

5. Schneider, K.M., "TOPEX/Poseidon Project Post-Expanded Checker Software Specification Document." JPL Internal Report, October 1992.